CAPITOLUL 1. DESPRE ABORDAREA OBIECTUALĂ

1.1. *Introducere în universul obiectelor*

Programul (informatic) este definit în mod tradițional ca fiind transpunerea într-un limbaj de programare a *algoritmului* de rezolvare a unei probleme.

Se spune pe bună dreptate că, în terminologie informatică, nimic nu este mai uşor şi totodată mai greu de definit decât noţiunea de *obiect*.

O definiție aproximativă pentru *obiect* ar fi aceea de *entitate* cu o existență concretă sau abstractă, caracterizată printr-un *identificator* (nume, cod sau ceva similar), o serie de *proprietăți* şi care execută sau suportă anumite *acțiuni*.

Ce legătură au obiectele din lumea reală cu programele informatice? Ei bine, orice program încearcă să soluționeze o problemă "decupată" din realitate, deci implicând sub anumite forme obiecte.

Orientarea-obiect se poate defini ca fiind modelarea software-ului prin tehnicile de dezvoltare care facilitează construirea de sisteme complexe din componente individuale. Dintre stilurile de programare contemporane, ea favorizează cel mai mult apropierea de lumea reală.

Dintr-o perspectivă istorică, geneza şi evoluția limbajelor de programare, ca şi a tehnicilor de programare s-a legat aproape întotdeauna de evoluția sistemelor de calcul. Se cunosc mai multe generații de limbaje de programare, pornind de la cele de nivel scăzut (cod-maşină) şi ajungând la limbajele de generația a patra. La început (adică timp de vreo două decenii şi jumătate) conceperea de programe era apanajul specialiştilor (care trebuiau să cunoască amănunțit hardware-ul pe care lucrau). Pe măsură însă ce s-au construit calculatoare mai puternice şi mai accesibile (din punct de vedere al prețului), limbajele de programare s-au orientat mai mult către utilizatorul obişnuit şi activitatea de programare a putut fi abordată de un cerc mai larg de persoane. De exemplu, prima versiune a limbajului BASIC (**B**eginner's **A**llpurpose **S**ymbolic **I**nstruction **C**ode) a fost scrisă de Kurz şi Kemeny, de la Dartmouth College, SUA, în 1965 pentru familiarizarea studenților cu lucrul la calculator. Această versiune cuprindea doar 12 instrucțiuni (FORTRAN sau COBOL, utilizate în aplicații "serioase" erau mult mai complicate) care erau cuvinte din limba engleză (INPUT, PRINT etc.). Problema accesului la calculator era însă una spinoasă, timpul de prelucrare a unui mainframe din acea vreme fiind costisitor (se folosea sistemul *time-sharing*).

Două sunt paradigmele sau stilurile de programare care au marcat evoluția programării calculatoarelor ca fenomen: *paradigma procedurală* şi cea *declarativă* (figura 1).

La sfârşitul anilor '60 s-a instaurat aşa-zisa "criză a programării"[1]: multitudinea situațiilor "de pe teren" solicitau programatori tot mai mulți şi tot mai bine pregătiți, precum şi multe sute de ore-om de muncă. Criza aceasta persistă şi în zilele noastre, deşi s-au căutat multe soluții pentru depăşirea ei.

În domeniul *tehnicilor de programare*, un loc important îl ocupă încă *programarea structurată*, printre ai cărei promotori îi cităm pe E. Djikstra (cunoscut în anii 70 şi prin

[1] Recunoscută oficial la conferința NATO privind proiectarea software-ului, Garmisch, Germania, 1968.

înfierările aduse limbajelor de generaţia a treia[2]), O. Dahl, F. Knuth. Programarea structurată se bazează pe principiul "divide et impera", pe separarea activităţii de proiectare a algoritmilor de codarea lor, a introdus structurile de control[3] şi a reprezentat un pas însemnat în contracararea "crizei programării". Curente mai noi includ *programarea bazată pe evenimente* (proprie mediilor cu interfaţă grafică – Windows, MacOS), *programarea vizuală* şi *programarea orientată pe obiecte*. În cele mai multe dintre mediile moderne de dezvoltare a aplicaţiilor (Visual Basic, Delphi, Visual C++ ş.a.), coexistă elemente din programarea obiectuală, cea vizuală şi cea "evenimenţială".

Majoritatea conceptelor care se regăsesc astăzi în programarea orientată obiect au fost inventate în '60, însă limbajele orientate obiect au ajuns în prim plan în anii '80, când au avut loc două evenimente semnificative: publicarea în revista *Byte* (august 1981) a unui articol cu largă audienţă descriind limbajul Smalltalk şi prima conferinţă având în vedere limbajele de programare orientate obiect în Portland, Oregon – 1986.

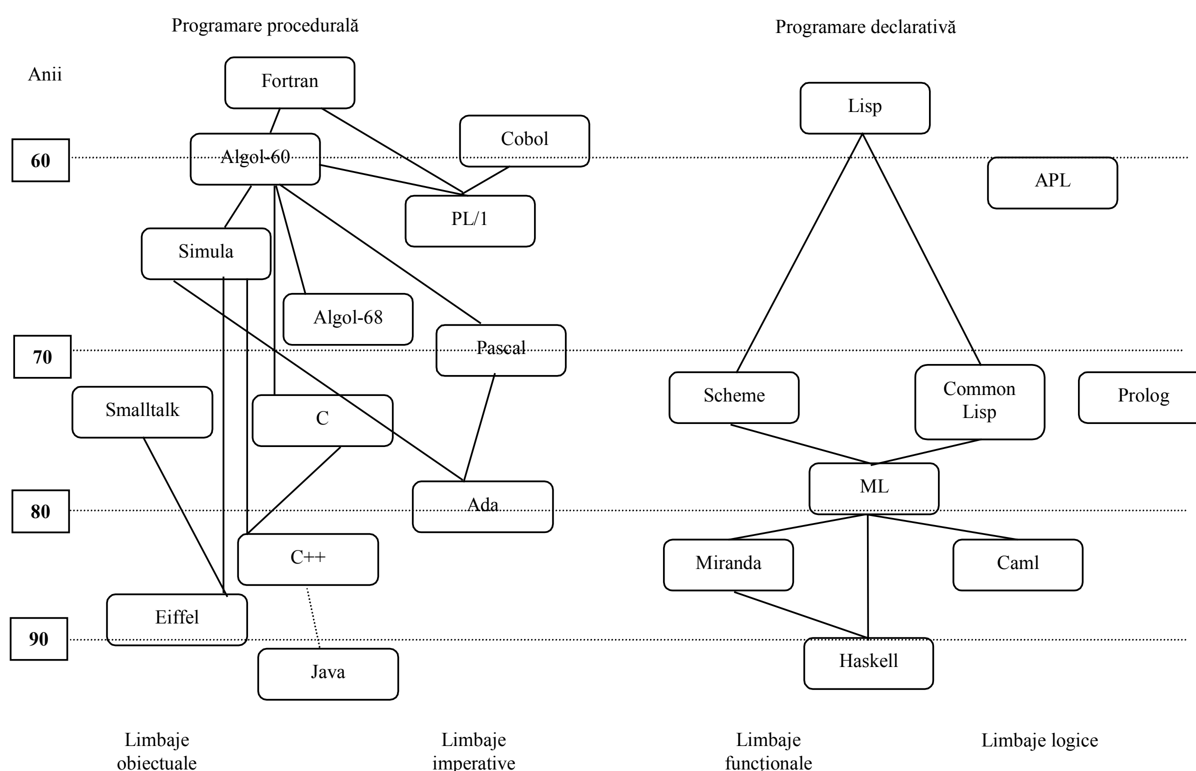


Fig. 1 – Evoluţia limbajelor de programare (după Giumale, C., *Limbajele de programare la sfârşit de secol, o barieră de cultură profesională?*, în PCReport nr. 8/1997)

Toate limbajele de programare dau posibilitatea *abstractizării*: limbajul de asamblare este o "uşoară" abstractizare a maşinii pentru care este proiectat. În programarea "clasică" (folosind limbajele de generaţia a treia, mai precis cele imperative), abstractizarea se întâlneşte pe larg[4], dar ea implică totuşi judecarea problemei în termeni de structură a calculatorului, adică ţinând

[2] "erezia programării cu GO TO"

[3] teorema lui Boehm şi Jacopini: orice program poate fi realizat prin combinarea *structurilor de control*: secvenţa, selecţia şi iteraţia, iar fiecare structură de control are *o singură intrare* şi *o singură ieşire*.

[4] FORTRAN, primul limbaj de nivel înalt, a introdus primul mecanism de abstractizare, prin *subrutine* şi *funcţii* (utilizatorul-neprogramator ştie *ce* face subrutina/funcţia, nu *cum* face).

cont de arhitectura von Neumann (vom numi în continuare calculatorul "spaţiul soluţiilor", în opoziţie cu lumea reală, care este "spaţiul problemei"). Rolul programatorului, simbolizat în figura 2, este de a realiza o "mapare" cât mai fidelă a spaţiului soluţiilor pe spaţiul problemei, operaţie care nu întotdeauna reprezintă un succes total, ea depinzând de flexibilitatea limbajului folosit şi de abilitatea programatorului. De aceea, programele–sursă sunt deseori dificil de înţeles, greu de întreţinut şi de modificat ulterior.

O alternativă la cele de mai sus o reprezintă *modelarea problemei* de rezolvat astfel încât "plierea" pe structura sistemului de calcul să fie mai facilă. Din acest punct de vedere, limbajele inteligenţei artificiale se comportă mai bine: LISP reduce toate problemele la liste, APL la algoritmi, iar PROLOG la lanţuri de decizie. Nici una dintre aceste soluţii nu are însă caracter de universalitate, deoarece majoritatea limbajelor de programare se orientează spre rezolvarea unor tipuri "specifice" de probleme.

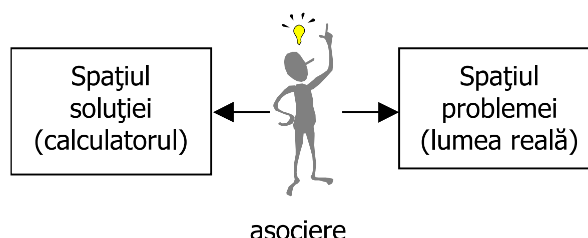


Fig. 2. Dilema programatorului

Drumul spre programarea orientată pe obiecte am putea spune că a presupus parcurgerea mai multor etape, bazate în fapt pe evoluţia modului de abstractizare a „spaţiului problemei":

1. *Programarea procedurală* în care abstractizarea presupunea elaborarea funcţiilor sub formă de proceduri;
2. *Programare modulară (Modula-2, ADA)* care a deschis calea abstractizării datelor. Modulele furnizau o sintactică pentru descompunerea programelor în componente mai mult sau mai puţin independente. În acest context, pentru a exprima semantica mai bine semantica unui modul a fost introdul conceptul de *tip de date abstract*, care oferea un set de servicii complet pentru datele *încapsulate*, având ca sarcină şi păstrarea integrităţii acestora.
3. *Programare orientată pe obiecte* care oferea o cale superioară de organizare a tipurilor, oferind şi posibilităţi reutilizare îmbunătăţită a acestora. În fapt conceptele deosebite introduse de POO înseamnă *polimorfism şi moştenire*.

Prin urmare abordarea OO face un pas înainte, oferind programatorului instrumente pentru reprezentarea elementelor din spaţiul problemei într-un mod cât mai natural. Această reprezentare este suficient de generală pentru ca programatorul să nu fie limitat la un anumit tip de problemă. Elementele din "spaţiul problemei" sunt referite ca "obiecte". Programul poate să se adapteze la complexitatea problemei adăugând noi (tipuri de) obiecte. Descrierea problemei se face în *termenii problemei şi nu în termenii soluţiei*: se specifică mai mult *ce* decât *cum* trebuie făcut.

1.2. Caracteristicile de bază ale obiectelor şi lucrului cu obiecte

Alan Kay, considerat după unii părintele programării orientate obiect, a identificat următoarele caracteristici ca fundamentale în POO:

1. "Orice" poate fi considerat obiect – *Everything is an object.*
2. Prelucrările sunt efectuate de către obiecte comunicând între ele: unele cerând altora să execute anumite acţiuni. Obiectele comunică trimiţând şi recepţionând *mesaje*. Un mesaj este o cerere pentru o acţiune însoţit eventual de anumite argumente care ar putea fi necesare pentru a duce la bun sfârşit o sarcină.
3. Fiecare obiect are propria sa *memorie*, care constă din alte obiecte.
4. Fiecare obiect este o *instanţă* a unei *clase*. O clasă grupează obiectele similare.
5. Clasa este "repository"-ul comportamentului asociat unui obiect. Adică, toate obiectele care sunt instanţe ale aceleaşi clase pot efectua aceleaşi acţiuni.
6. Clasele sunt organizate într-o structură ierarhică arborescentă (de regulă cu o singură rădăcină) numită *ierarhie de moştenire*. Memoria şi comportamentul asociat cu instanţele unei clase este pus automat la dipoziţia oricărei clase descendentă în structura arborescentă.

Prin urmare caracteristicile de bază ale orientării-obiect sunt:

- paradigma obiect-mesaj;
- tipurile abstracte de date;
- moştenirea;
- identitatea obiectelor.

1.2.1. Paradigma obiect-mesaj

Universul computaţional al acestei paradigme este construit din *obiecte*; fiecare obiect răspunde la o colecţie determinată de *mesaje*.

În programarea procedurală, rutinele sau funcţiile se apelează unele pe altele pentru a returna anumite *date* sau pentru a modifica nişte parametri de intrare. În modelul obiect-mesaj, în loc de data elementară avem un *obiect* capabil să proceseze *cereri* cunoscute ca *mesaje*. Aceste mesaje:

- pot cere obiectului să efectueze "ceva" şi să returneze un rezultat;
- pot modifica starea obiectului (adică pot modifica valorile unor proprietăţi ale obiectului).

Acţiunea (execuţia) în programarea orientată-obiect este iniţiată prin transmitarea unui *mesaj* către un *agent* (un *obiect*) responsabil de efectuarea respectivei acţiuni. Mesajul codifică cererea unei acţiuni şi este însoţit de informaţii adiţionale (*argumente*) necesare pentru a duce la bun sfârşit acţiunea indicată. *Receptorul* este obiectul către care este trimis mesajul. Dacă receptorul acceptă mesajul, acceptă implicit şi responsabilitatea de a efectua acţiunea indicată. Ca răspuns la mesaj receptorul va executa o anumită cerere pentru a satisface cererea.

Un principiu esenţial al acestei paradigme este *ascunderea informaţiei* vizavi de transmiterea mesajelor, cu alte cuvinte clientul care trimite cererea nu are nevoie (şi uneori nici voie) să cunoască mijloacele efective prin care este onorată cererea.

Din perspectiva abstractizării *ascunderea informației* (information hiding) reprezintă omisiunea intenționată a detaliilor în dezvoltarea reprezentării unei abstracțiuni.

Principiul ascunderii informaţiei se regăseşte şi în programarea prin limbajele convenționale (programarea structurată) sub forma apelurilor de proceduri (sau funcţii). Există însă două diferențe: în primul rând apelul unei proceduri nu desemnează în mod explicit un receptor, iar în al doilea rând în programarea orientată-obiect interpretarea mesajului (metoda folosită pentru a răspunde cererii transmise) este determinată de receptor putând varia astfel prin raportarea la receptori diferiți. De obicei, receptorul efectiv al unui mesaj dat nu este cunoscut decât în momentul execuției (la runtime), aşa încât determinarea metodei care trebui invocată nu este conoscută până la acel moment. Această situație este cunoscută sub numele de *legarea întârziată* (late binding) între mesaj (numele funcţiei sau procedurii) şi fragmentul de cod (metoda) folosită pentru a răspunde cererii exprimate prin mesajul dat, în contradicție cu *legarea în momentul compilării* (early-binding sau compile-time binding sau link-binding) a numelui cu fragmentul de cod din apelul clasic de proceduri.

1.2.2. Tipuri abstracte de date

Multe limbaje (chiar procedurale) dau posibilitatea utilizării unor tipuri de date definite de utilizator[5]. Dar programatorii au simțit nevoia introducerii unor noi tipuri, care să înglobeze pe lângă valori (care dau starea) şi comportamentul şi să ofere o interfață pentru accesarea stării şi comportamentului. Faptul că utilizatorul se serveşte exclusiv de această interfață şi percepe tipul de dată respectiv ca pe o “cutie neagră" a sugerat termenul de *tip abstract de dată* (pe scurt, TAD[6]).

Tipurile abstracte de date definesc atât *structura* obiectului (aparența), cât şi comportamentul său (*care mesaje îi sunt aplicabile*). Se ascunde astfel reprezentarea internă a obiectului de mediul înconjurător şi se protejează algoritmii interni care implementează comportamentul obiectului[7]. Termenul de “ascunderea informației” este cunoscut în teorie sub denumirea de *încapsulare*. Prin urmare principiul ascunderii informației oferă posibilitatea obținerii unor abstracțiuni cu *stare şi comportament autonom*.

Limbajele care implementează tipurile abstracte de date oferă construcții pentru definirea *structurilor de date* şi totodată a *operațiilor* utilizate pentru manipularea *instanțelor* acestor structuri de date. În plus, manipularea instanțelor unui anumit tip are loc *numai* prin operațiile asociate tipului respectiv.

Exemple de TAD: modulul (limbajul Modula-2), clusterul (limbajul CLU), pachetul (limbajul Ada), clasa (C++, Smalltalk, FoxPro), modulul-clasă (Visual Basic).

1.2.2.1. Interfață şi implementare

În domeniul dezvoltării aplicațiilor software, pentru a desemna distincția dintre aspectele funcționale (*ce* trebuie făcut, *ce* se poate obține sau *ce* se aşteaptă ca rezultat) şi apectele

---

[5] În BASIC putem crea un tip de dată definit de utilizator cu *TYPE…END TYPE*, iar în C cu *struct*.

[6] În engleză, ADT (de la Abstract Data Type).

[7] Algoritmii respectivi pot fi chiar rescrişi ulterior pentru a eficientiza codul. Interfața însă nu mai trebuie modificată dacă deja folosim respectivul TAD într-un program.

procedurale (*cum* se realizează) ale unei sarcini anume se folosesc termenii *interfaţă* şi *implementare*. O interfaţă descrie ce este desemnat să facă un sistem. Aceasta reprezintă viziunea externă, viziunea pe care trebuie să o înţeleagă utilizatorii. Interfaţa nu precizează nimic cu privire la modul în care ar trebui efectuată respectiva sarcină. Aşa încât pentru a-şi dobândi funcţionalitatea desemnată (descriptiv), o interfaţă trebuie să-şi găsească o *implementare* corespunzătoare care întregeşte abstracţiunea luată în considerare.

Distincţia dintre interfaţă şi implementare nu numai că simplifică înţelegerea unei probleme la un nivel de abstractizare mai înalt (din moment ce specificaţia descriptivă a unei interfeţe este mult mai simplă decât descrierea oricărei implementări particulare) ci deschide şi posibilitatea interschimbării componentelor software.

Faptul că interfeţele descriu serviciile oferite de o componentă software fără a descrie tehnicile folosite pentru implementarea lor conduce la formalizarea unei *viziuni a serviciilor* în descrierea sistemelor.

1.2.2.2. Clase

Clasa este conceptul cel mai des utilizat pentru implementarea tipurilor abstracte de date în limbajele OO.

Aristotel a fost probabil primul care a început un studiu serios al noţiunii de *clasă* sau *tip*. El vorbea despre "clasa peştilor" şi "clasa păsărilor", pentru a încadra obiecte unice în categorii de obiecte care au caracteristici comune.

Cuvintele *clasă* şi *tip* sunt adesea folosite ca sinonime.

O definiţie minimală a clasei include:

- *numele* clasei;
- operaţiile externe pentru manipularea instanţelor clasei (adică metodele) - referite ca *interfaţă*, plus codul care implementează metodele clasei + într-un cuvânt, *implementare* (din exterior, metodele sunt accesibile prin *semnătura* lor);
- reprezentarea internă, care păstrează valorile diferitelor stări ale instanţelor unei clase, cu alte cuvinte atributele sau proprietăţile; valorile acestora variază pentru fiecare instanţă a clasei.

Iată o reprezentare în pseudocod:

```
clasa ContDeActiv
variabile:
Simbol
Denumire
Sold
metode:
CalculSoldNou()
metoda CalculSoldNou (Sold, RulajD, RulajC)
    Sold=Sold+RulajD-RulajC
```

Orice clasă trebuie să ofere un mecanism de creare a unor noi instanţe (obiecte) şi de distrugere a celor existente şi nefolosite (constructori şi destructori de obiecte).

Deşi valorile variabilelor pot diferi de la o instanţă a clasei la alta, *toate instanţele partajează codul care implementează interfaţa*. În concordanţă cu mecanismul încapsulării,

variabilele de instanţă *private* nu pot fi accesate decât prin intermediul unor *funcţii* explicit declarate ca *publice*.

1.2.2.3. Compunerea

Compunerea este o tehnică esenţială în crearea unor structuri complexe pornind de la părţi simple. Ideea este să se înceapă de la câteva forme primitive după care să se formuleze regulile după care formele existente să se combine pentru a obţine noi forme. Cheia este ca prin compoziţie să se poată revalorifica oricând atât noile forme obţinute până la un moment dat cât şi formele primitive originale. Un exemplu classic al utilizării compoziţiei este modul în care prin intermediul bibliotecilor intefeţelor grafice se obţine modul de afişare a ferestrelor.

Declarea câmpurilor/membrilor/variabilelor din definiţia unei clase ca fiind obiecte de tipuri implementate de alte clase nu desemnează altceva decât un mecanism de compunere.

1.2.2.4. Containere şi extensii ale claselor

Extensia unei clase corespunde tuturor obiectelor din acea clasă care au fost create şi nu au fost (încă) distruse. Cunoaşterea extensiei îşi dovedeşte utilitatea în cazul SGBD-urilor care prelucrează mari cantităţi de date.

Un *container* poate fi definit ca o clasă generică, ale cărei instanţe sunt colecţii de obiecte, manipulabile prin operaţii asociate. Pot fi subclase precum *Set*, *Bag* (multiset) sau *List*, fiecare cu metode specifice de acces, ca de exemplu reuniunea seturilor sau enumerarea obiectelor din colecţie.

Putem sumariza astfel **avantajele tipurilor abstracte** de date:

- mai bună conceptualizare şi modelare a lumii reale; creşterea capacităţii de reprezentare şi de înţelegere; categorizarea obiectelor pornind de la structuri şi trăsături de comportament comune;
- creşterea robusteţii sistemului informatic. Dacă limbajul de programare este tipizat, se poate evita apariţia unor erori la execuţie. Restricţiile de integritate impuse datelor şi operaţiilor ajută la menţinerea consistenţei datelor şi corectitudinii programelor;
- sporirea performanţei, prin optimizări în momentul compilării;
- mai bună reflectare a semanticii datelor;
- separarea *implementării* de specificaţie (*interfaţă*) – modificarea implementării *nu va afecta* interfaţa publică a tipului abstract de dată;
- posibilitatea de a realiza sisteme extensibile, prin intermediul reutilizării.

1.2.3. Moştenirea

Pe lângă modelarea cât mai fidelă a realităţii, POO tinde de asemenea să realizeze extensibilitatea şi reutilizabilitatea software-ului. Tehnicile prin care acestea sunt posibile sunt *compunerea* (definită mai înainte) şi *moştenirea*. Moştenirea permite construcţia de noi obiecte sau module de program – de exemplu, a unor clase – pe baza unei ierarhii de module existente. Aceasta elimină nevoia reproiectării şi rescrierii de programe “de la zero”.

Prin urmare moştenirea oferă posibilitatea organizării abstracţiunilor (tipurilor) existente în *ierarhii de clase specializate incremental*.

Din punctul de vedere al abstractizării compunerea şi moştenirea au efecte similare: se elimină anumite detalii, concentrarea rămând asupra unor mai puține caracteristici care trebui implementate la nivelul următor.

Diferența dintre compunere şi moştenire rezidă din faptul că nivelul de abstractizare mai specializat este reprezentativ pentru nivelul de abstractizare mai general în cazul moştenirii. Adică printre caracteristicile nivelului specializat se regăsesc şi toate caracteristicile nivelului mai general (excepțiile desemnând *suprascrierile* unor trăsături din clasele de bază). Prin compunere se împrumută numai funcționalitate, prin moştenire poate împrumuta eventual şi implementarea.

Noile clase pot moşteni atât *comportamentul* (metode, funcții) cât şi *reprezentarea* (variabile, atribute) de la clasele existente.

Moştenirea îşi are rădăcinile în cercetările legate de inteligența artificială. În 1968, Ross Quillian a prezentat modelul teoretic al memoriei asociative (unul dintre primele modele de rețea semantică). O rețea semantică se compune din noduri care reprezintă obiecte şi din legături reprezentând relații (de tip **is-a** şi **has-a**).

În limbajele de programare OO vorbim despre moştenirea de clasă (se creează o clasă derivată pe seama unei/unor clase părinte sau superclase ori clase de bază – cu alte cuvinte, prin specializare).

Ex.: clasa de bază este comunitatea academică, iar clasele derivate sunt studenti şi profesori.

Să reținem că moştenirea are două aspecte: structural şi comportamental.

Avantaje ale moştenirii:

- oferă un model natural pentru organizarea informației);
- permite partajarea codului (implementării) şi interfeței;
- permite claselor să fie definite pe baza ierarhiilor existente şi nu de la zero.

1.2.4. Polimorfismul

Polimorfismul este un alt concept-cheie al POO, în strânsă relație cu moştenirea şi cu ierarhiile de clase. A fost definit inițial de Christopher Strachey (1967).

Polimorfismul se defineşte ca fiind capacitatea unei metode de a se aplica obiectelor din diferite clase.

Polimorfismul ad-hoc se referă la faptul că idetificatorul unei proceduri sau metode desemnează mai mul decât o singură procedură (fragment de cod).

Polimorfism pur se referă la caracteristica unei funcții singulare de a putea fi executată cu argumente de tipuri de diferite.

În practică se simte adesea necesitatea redefinirii unei metode pentru a-i da o implementare specifică pentru clasa derivată. Această implementare va ține cont şi de proprietățile locale (nemoştenite) ale clasei. Deci operația redefinită (*overridden*) va avea acelaşi

nume cu cea din clasa de bază, dar implementare diferită în clasa derivată. Despre redefinire sau suprascriere am vorbit mai sus, iar unii autori numesc suprascrierea *polimorfism* ad-hoc.

Prin mecanismul de *legare întârziată* (late binding), la momentul execuției se alege corpul (implementarea) adecvat(ă) metodei. În programul-client, în etapa codării (scrierii programului), este suficientă invocarea numelui metodei. Prin aceasta se elimină nevoia modificării programelor utilizatoare de obiecte, în cazul apariției de clase derivate.

Legarea întârziată reprezintă mecanismul prin care se pot adăuga noi tipuri în ierarhia de clase, fără a modifica programele de bază în care pot fi implicate în prelucrări.

După cum am precizat la un moment dat, în programarea orientată-obiect *interpretarea* mesajelor se poate realiza diferit funcțiile de tipurile diferite de obiect care ar putea recepționa acel mesaj. Prin urmare, comportamentul rezultat şi răspunsul obținut depind de obiectul receptor. Legarea aceastei caracteristică de mecanismul de moştenire duce la situații în care aceeaşi metodă (definită inițial pentru un anume tip, cu anumite argumente) va conduce la rezultate diferite. Prin moştenire o aceeaşi metodă va fi disponibilă tuturor claselor care derivă din clasa în care aceasta a fost definită inițial. Însă funcție de natura lor, anumite subclase pot rescrie metoda respectivă ca implementare, lăsând însă nemodificată *semnătura* ei (restricția de moştenire). Prin urmare acelaşi mesaj poate determina funcție de obiectul care-l recepționează comportamente distincte. Astfel de comportamente se numesc *polimorfice*.

Avantajele polimorfismului sunt evidente şi reprezintă un alt atu al programării obiectuale față de programarea clasică (procedurală). Ele pot fi exemplificate în tabelul de mai jos, în care, în pseudocod, este descris procesul de calcul al salariului unui angajat. S-a considerat că algoritmul de calcul este specific funcției angajatului respectiv.

1.2.5. Identitatea obiectelor

Dacă *tipurile abstracte de date şi mecanismul moştenirii* modelează şi organizează obiectele din punct de vedere al tipurilor sau claselor, *identitatea obiectelor* le organizează în spațiul manipulat de un program OO.

Identitatea este proprietatea unui obiect de a se distinge de toate celelalte. Cea mai uzitată tehnică de identificare a obiectelor în limbajele de programare este de a le înzestra cu nume definite de utilizator.

În programarea "clasică" nu există suport pentru identitatea obiectelor, atât timp cât se poate face referire la variabile fără ca acestea să fie legate de obiecte (putem invoca o variabilă *strNume* sau o variabilă *dDataNasterii*). În acest caz trebuie specificat dacă ele (variabilele) se referă sau nu la acelaşi obiect.

În limbajele OO, acest dezavantaj nu mai apare, datorită introducerii operatorului "= =", care testează identitatea a 2 obiecte şi nu egalitatea din punct de vedere al conținutului, ca în cazul operatorului "=".

Avantaje ale identității obiectelor:

- se asigură reprezentarea directă a unor spații de obiecte;
- nu se pune problema menținerii restricției de integritate referențială;

- diferitele operații asociate cu identitatea obiectelor oferă funcționalități avansate pentru manipularea obiectelor.